Efficient Safety Enforcement for Maude Programs via Program Specialization in the ÁTAME system

M. Alpuente, D. Ballis and J. Sapiña

Abstract. Program specialization is mainly recognized as a powerful technique for optimizing software systems. Nonetheless, it can also be productively employed in other application areas.

This paper presents an assertion-guided program specialization methodology for efficiently imposing safety properties on software systems. The program specializer takes as input a set $\mathcal{A}$ of logical assertions that specifies the expected system behavior plus a software system that is modeled as a Maude program $\mathcal{R}$ that may violate some of the assertions in $\mathcal{A}$. The outcome is a safe refinement $\mathcal{R}^{\triangleright}$ of $\mathcal{R}$ in which every system computation is a good run of $\mathcal{R}$, i.e., it satisfies the assertions in $\mathcal{A}$. The specialization technique has been fully automated in the ÁTAME system and ensures that no good run of $\mathcal{R}$ is removed from $\mathcal{R}^{\triangleright}$, while the number of bad runs is reduced to zero. The efficiency and scalability of our technique is empirically demonstrated by means of a thorough experimental evaluation of the ÁTAME system, which shows fast specialization times and good performance of the computed specializations, even for large assertion sets.

Mathematics Subject Classification (2010). Primary 68N30.

Keywords. Safety properties, Assertions, Program Transformation, Maude.

1. Introduction

Program specialization is the mechanized task of producing a refined executable version of an overly general input program that meets the user intent expressed by means of some sort of specification (e.g., logical assertions, functional specifications, reference implementations, examples, contracts, passing and failing tests). In the literature, program specialization is often used to mean partial evaluation [9] —a program optimization technique that transforms a program with n inputs into a simpler and typically faster version where some of the inputs are fixed to specific values.

However, when we pursue program safety rather than program optimization, it is often useful to consider a kind of dual notion of specialization where a program P of n outputs (or more generally, a program that explores n program computations) is transformed into a more specific version of P in which some of the output computations are disregarded according to a formal specification that rigorously formalizes forbidden and/or allowed program behaviors. This way, we guarantee that all of the runs of the specialized program are safe in the sense that they cannot reach bad states (e.g., system-critical states, deadlock states). Safe program behaviors can be elegantly specified by means of assertions, that is, logical statements that can virtually characterize any set of program states in a purely declarative way [2].

Maude [7] is a high-level programming language and system that supports functional, concurrent, logic, and object-oriented computations and provides equational rewriting and reasoning modulo algebraic axioms such as associativity, commutativity, and identity. Maude is well-suited to specifying software systems. In fact, a Maude program is essentially a rewrite theory $\mathcal{R} = (\Sigma, E, R)$ where E is a canonical (i.e., confluent and terminating) equational theory that models system states as terms of an algebraic data type, and R is a set of rewrite rules that define transitions between states, i.e., the system's dynamics that might encompass nondeterminism, concurrency, or even nontermination. Both equations in E and rewrite rules in R are built upon the operators of a typed signature Σ.

In this paper, we propose a safety enforcement technique for Maude programs that combines program specialization with assertional constraints. More precisely, our specialization technique works with Maude programs that are equipped with system assertions, with each assertion consisting of a pair $\Pi \mid \varphi$, where Π (the *state template*) is a term and φ (the *state invariant*) is a quantifier-free first-order formula with equality that defines a safety property φ which must be enforced on all the system states that match (modulo equations and axioms) the state template Π. In our approach, assertions take an active role since they are directly embedded into the specialized program to safely guide its execution. Given a set of system assertions $\mathcal{A}$ and an overly general Maude program $\mathcal{R} = (\Sigma, E, R)$ (i.e., a program that deploys all desired computations but may disprove some of the assertions), our transformation coerces $\mathcal{R}$ into a specialized program $\mathcal{R}^{\triangleright}$ that enforces $\mathcal{A}$. This means that: (i) every execution of $\mathcal{R}^{\triangleright}$ is an execution of $\mathcal{R}$ (i.e., no spurious computation states are produced); and (ii) every assertion in $\mathcal{A}$ is satisfied by all computation states in $\mathcal{R}^{\triangleright}$. The program $\mathcal{R}^{\triangleright}$ is obtained from $\mathcal{R}$ by inserting suitable conditions (abetted by the assertions of $\mathcal{A}$) in the rules of R and defining them by means of new equations that are added to E until a suitable adaptation of the original program is automatically inferred which satisfies all the assertions.

The advantage of this technique is that more refined versions of a program can be incrementally built without any programming effort by simply adding new logical constraints into the given assertion set. Specifically, this makes it possible to adapt existing Maude programs to predefined safety policies and allows the inexperienced user to largely forget about Maude syntax and semantics.

This article is a revised and extended version of our previous work [3], whose main contributions can be summarized as follows.

1. Further conceptual and technical content has been included to make the article self-contained. Specifically, a more in-depth description of the specialization technique has been provided together with various examples that illustrate each step of the program transformation.
2. The ÁTAME tool that implements our specialization technique has been endowed with an additional facility that allows the computation space of the specialized program to be searched. This feature is especially useful when the user wants to individualize a particular safe computation, which specifies a deterministic sequence of actions, within the computation space of a (possibly) nondeterministic Maude program.
3. A solid, empirical evaluation of the ÁTAME system has been conducted to show the efficiency and scalability of the proposed approach. Program safety is generally achieved by means of dynamic verification techniques that execute programs in a monitored runtime environment to detect assertion violations. In contrast, our specialization approach is static and yields a safe program by construction that can be executed in a standard runtime environment.

Plan of the paper. After some technical preliminaries in Section 2, we introduce a Maude program that models a nondeterministic dam controller that will be used as a running example to illustrate the kind of specialization that we aim to produce automatically. Section 3 shows how safety policies can actually be defined as system assertions in our rewriting setting and then applied for program specialization. Section 4 explains how safety enforcement is achieved in the ÁTAME system, which implements our specialization methodology. Also, this section describes two additional features of our system that are well-suited to both inspecting the runtime behavior of the computed specialization and extracting selected computations from the specialization computation space. Section 5 presents a complete, step-by-step ÁTAME specialization session that describes all of the main features of the system. In Section 6, the specialization technique is empirically evaluated on a large set of benchmarks that consider different kinds of software systems (e.g. network protocols, web applications, mutual exclusion algorithms) together with safety policies of increasing size. Section 7 reviews the related work and presents our conclusions.

2. Software Systems as Maude Programs

Nondeterministic as well as concurrent software systems can be formalized through Maude programs. A Maude program is essentially a rewrite theory (Σ, E, R) where Σ is a signature that contains the program operators together with their type definition, E is a canonical (membership) equational theory that models system states as terms of an algebraic data type, and R is a set of rewrite rules that define transitions between states. Algebraic structures often involve axioms like associativity (`assoc`), commutativity (`comm`), and/or identity (also known as unity) (`id`) of function symbols, which cannot be handled by ordinary term rewriting but instead are handled implicitly by working with congruence classes of terms. More precisely, the membership equational theory E is decomposed into a disjoint union $E = \Delta \uplus Ax$. Here, the set Δ consists of (conditional) equations and membership axioms (i.e., axioms that assert the type or *sort* of some terms) that are implicitly oriented from left to right as rewrite rules (and operationally used as simplification rules), and Ax is a set of algebraic axioms that are implicitly expressed as function attributes and are only used for Ax-matching.

The system evolves by rewriting states using *equational rewriting*, i.e., rewriting with the rewrite rules in R *modulo* the equations and axioms in E [13]. Formally, system computations (also called execution traces) correspond to rewrite sequences $t_0 \xrightarrow{r_0}_E t_1 \xrightarrow{r_1}_E \ldots$, where $t \xrightarrow{r}_E t'$ denotes a transition (modulo E) from state t to t' via the rewrite rule of R that is uniquely labeled with label r. The transition space of all computations in $\mathcal{R}$ from the initial state t_0 can be represented as a *computation tree* whose branches specify all of the system computations in $\mathcal{R}$ that originate from t_0.

The following Maude program will be used as a running example throughout the paper.

Example 1. *Consider a Maude program $\mathcal{R}_{\texttt{DAM}}$ that models a simplified, nondeterministic dam controlling system to monitor and manage the water volume of a given basin*[1]*. In the program code, variable names are fully capitalized.*

We assume that the dam is provided with three spillways called `s1`*,* `s2`*, and* `s3`*, each of which has four possible aperture widths of increasing discharge capacity* `close`*,* `open1`*,* `open2`*,* `open3`*. Each spillway is formally specified by a term* `[S,O]`*, where* $\texttt{S} \in \{\texttt{s1}, \texttt{s2}, \texttt{s3}\}$ *and* $\texttt{O} \in \{\texttt{close}, \texttt{open1}, \texttt{open2}, \texttt{open3}\}$*. A global spillway configuration* `[s1,O1] [s2,O2] [s3,O3]` *is a multiset that groups the three spillways together by means of the usual associative and commutative infix, union operator* _ _ *(written in mixfix notation with empty syntax) whose identity is the constant* `empty`*. System states are defined by terms of the form* { `SC | V | T | AC` } *where* `SC` *is a global spillway configuration,* `V` *is a rational number that indicates the basin water volume (in* m^3*),* `T` *is a natural number that timestamps the current configuration, and* `AC` *is a Boolean flag, called* `apertureCommand`*, that enables changes of the spillway aperture widths only when its value is* `true`*.*

Figure 1 *shows the equational specification that formalizes basin water inflow and outflow. To keep the exposition simple, we assume that the basin water inflow is constant, while the basin*

[1] Maude's syntax is hopefully self-explanatory. Due to space limitations and for the sake of clarity, we only highlight those details of the system that are relevant to this work. A complete Maude specification of the dam controller is available at the ÁTAME website at `http://safe-tools.dsic.upv.es/atame`. For more information about the Maude language, see [7].

```
eq inflow = 2000 .                --- Basin water inflow
eq aperture(close) = 0 .          --- Outflow for a closed spillway
eq aperture(open1) = 200 .        --- Outflow for aperture width open1
eq aperture(open2) = 400 .        --- Outflow for aperture width open2
eq aperture(open3) = 1200 .       --- Outflow for aperture width open3

--- Basin water outflow for a given spillway configuration
eq outflow(empty) = 0 .
eq outflow([S,O] SS) = aperture(O) + outflow(SS) .
```

FIGURE 1. Equational definition of basin inflow and outflow.

outflow depends on the width of the spillway apertures and can be computed as the sum of the outflows of each spillway in the spillway configuration. Note that inflow and outflow values are measured in m^3/min *and are hard-coded into the dam controller. More realistic scenarios could be easily defined by making the basin inflow and outflow functions more sophisticated.*

```
rl [nocmd] : { SC | V | T | true } => { SC | V | T | false } .
rl [openC-1] :
    { [S,close] SS | V | T | true } => { [S,open1] SS | V | T | false } .
rl [open1-2] :
    { [S,open1] SS | V | T | true } => { [S,open2] SS | V | T | false } .
rl [open2-3] :
    { [S,open2] SS | V | T | true } => { [S,open3] SS | V | T | false } .
rl [close1-C] :
    { [S,open1] SS | V | T | true } => { [S,close] SS | V | T | false } .
rl [close2-1] :
    { [S,open2] SS | V | T | true } => { [S,open1] SS | V | T | false } .
rl [close3-2] :
    { [S,open3] SS | V | T | true } => { [S,open2] SS | V | T | false } .
crl [volume] : { SC | V | T | false } => { SC | V′ | (T + deltaT) | true }
    if V′ := (V + inflow * deltaT) - (outflow(SC) * deltaT) .
```

FIGURE 2. (Conditional) rewrite rules for the dam controlling system.

The system dynamics is specified by the eight rewrite rules in Figure 2*, which implement system state transitions. The* `openX-Y` *rewrite rules progressively increment the aperture width of a given spillway (e.g., the rule* `open1-2` *increases the aperture of the spillway* `S` *from level* `open1` *to level* `open2`*). Dually,* `closeX-Y` *rewrite rules progressively decrease the aperture width of a spillway. The rule* `nocmd` *specifies the empty command, which basically states that no action is taken on the spillway configuration by the dam controller at time instant* `T`*. The rule is fired only when the* `AC` *flag is enabled, and its application disables the flag to allow a new basin water volume to be computed in the next time instant. These eight rules, called* aperture command *rules, implement instantaneous spillway modifications that do not change the time instant or the basin water volume.*

The temporal evolution of the basin water volume is specified by the conditional rewrite rule `volume` *that computes the volume* `V′` *at time* `T + deltaT`*, given the input volume* `V` *at time* `T`*. The parameter* `deltaT` *is measured in minutes and can be set by the user. The volume computation changes the input volume* `V` *by adding the water inflow and subtracting the corresponding water outflow over the* `deltaT` *interval.*

The use of the `apertureCommand` *flag in the rule definitions guarantees a fair interleaving between the applications of the rule* `volume` *and the remaining aperture command rules. Specifically, this implies that a new basin water volume is computed after each spillway aperture width modification.*

When convenient, we interpret the Maude program $\mathcal{R}_{\texttt{DAM}}$ as the rewrite theory $\mathcal{R}_{\texttt{DAM}} = (\Sigma_D, E_D, R_D)$, where Σ_D is the signature of the dam controller, E_D consists of the equations of Figure 1*, and R_D contains the rewrite rules in Figure* 2*.*

Note that computations in $\mathcal{R}_{\texttt{DAM}}$ may reach potentially hazardous system states (e.g., an extremely high water volume) as shown Example 2. This is because $\mathcal{R}_{\texttt{DAM}}$ does not implement any spillway management policy that safely restricts the applications of the aperture command rules.

Example 2. *Consider the rewrite theory $\mathcal{R}_{\texttt{DAM}} = (\Sigma_D, E_D, R_D)$ of Example* 1 *and assume a critical basin water volume threshold equal to* 50 *million cubic meters. Then, there exists the following unsafe computation in $\mathcal{R}_{\texttt{DAM}}$ that reaches a state exceeding the considered threshold.*

$$\begin{array}{ll}
\texttt{\{[s1,open1] [s2,close] [s3,close] | 49985000 | 0 | true\}} & \xrightarrow{\texttt{close1-C}}_{E_D} \\
\texttt{\{[s1,close] [s2,close] [s3,close] | 49980000 | 0 | false\}} & \xrightarrow{\texttt{volume}}_{E_D} \\
\texttt{\{[s1,close] [s2,close] [s3,close] | 49995000 | 5 | true\}} & \xrightarrow{\texttt{nocmd}}_{E_D} \\
\texttt{\{[s1,close] [s2,close] [s3,close] | 49995000 | 5 | false\}} & \xrightarrow{\texttt{volume}}_{E_D} \\
\texttt{\{[s1,close] [s2,close] [s3,close] | 50005000 | 10 | true\}} &
\end{array}$$

In the sequel, we describe how the unsafe program $\mathcal{R}_{\texttt{DAM}}$ can be statically refined by defining logical assertions that are used to enforce critical properties at specialization time.

3. Specifying Safety Policies via Assertions

A *safety policy* for a Maude program $\mathcal{R} = (\Sigma, E, R)$, with $E = \Delta \uplus Ax$, is defined by means of a set $\mathcal{A}$ of system assertions that $\mathcal{R}$ must satisfy, with each assertion being of the form $\Pi \,|\, \varphi$. Intuitively, assertions specify safe system states as follows. A system state s is safe w.r.t. an assertion $\Pi \mid \varphi$ iff, for every subterm of s that matches (modulo the axioms in Ax) the algebraic structure of Π with substitution σ, the constraints given by the instantiated invariant $\varphi\sigma$ are satisfied in E. Besides the usual Boolean operators and Maude predefined relational predicates, the state invariant φ may include user-defined predicates as well as functions that can be specified via suitable equational definitions.

Example 3. *Let us consider the user-defined function* `openSpillways(SC)` *that returns the number of open spillways in the spillway configuration* `SC`*, whose equational definition is*

```
eq openSpillways(empty) = 0 .
eq openSpillways([S,O] SC) = if (O =/= close)
                             then (1 + openSpillways(SC))
                             else openSpillways(SC) fi .
```

and the safety policy $\mathcal{A}_{\texttt{DAM}}$ of Figure 3 *for the dam controller of Example* 1 *that specifies some safety constraints to prevent critical basin situations.*

```
(a1) { SC | V | T | AC } | (V < 50000000)
(a2) { [ S1,O1 ] [ S2,O2 ] [ S3,O3 ] | V:Rat | T:TimeStamp | AC:Bool } |
                (V:Rat > 40000000) implies (
                    (O1 == open3 and O2 =/= close and O3 =/= close) or
                    (O2 == open3 and O1 =/= close and O3 =/= close) or
                    (O3 == open3 and O1 =/= close and O2 =/= close))
(a3) { SC | V | T | AC } | (V < 10000000) implies
                                (openSpillways(SC) == 0)
(a4) { SC | V | T | AC } | ((V >= 10000000) and (V <= 40000000)) implies
                                (openSpillways(SC) == 2)
```

FIGURE 3. Safety policy $\mathcal{A}_{\texttt{DAM}}$ for the dam controller $\mathcal{R}_{\texttt{DAM}}$.

More specifically, assertion `a1` *states that, in every system state, the basin water volume must be less than 50 million* m^3 *to avoid dam bursts and potentially disastrous floods. Assertion* `a2` *specifies that, whenever the basin water volume is greater than 40 million* m^3*, all of the spillways must be open and the aperture width of at least one spillway must be maximal (level* `open3`*). Assertion* `a3` *requires the closure of all the spillways when the basin water volume is particularly low (10 million* m^3*). Finally, assertion* `a4` *specifies the spillway handling for an intermediate water volume (10 million* $m^3 \leq \mathtt{V} \leq 40$ *million* m^3*); in this scenario we require that exactly two spillways be constantly open.*

4. Computing Safe Maude Programs with ÁTAME

Program specialization techniques make it possible to automatically transform a program into a specialized version, according to an execution context.

ÁTAME (*Assertion-based Theory Amendment in MaudE*) implements the program correction technique of [4], where we use assertions to set the specialization scenario and guide the refinement technique to transform a Maude program $\mathcal{R}$ into a specialized program $\mathcal{R}^{\triangleright}$ that complies with the safety policy $\mathcal{A}$. This is done in two phases as follows.

Phase 1. Compiling the assertion set $\mathcal{A}$ into an assertional (equational) theory. This phase translates the safety policy $\mathcal{A}$ to be fulfilled into an executable equational definition $Eq(\mathcal{A})$ that can be used to detect assertion violations within system states. Roughly speaking, given a system state s, a violation of some assertion in $\mathcal{A}$ is detected in s if s can be simplified into the special constant `fail` by using an extended equational theory that consists of the equational theory E of $\mathcal{R}$ augmented with $Eq(\mathcal{A})$.

Formally, for each assertion $\Pi|\varphi \in \mathcal{A}$, $Eq(\mathcal{A})$ includes an equation

$$\Pi\text{-ren} = \mathtt{fail} \text{ if } not(ori(\varphi))$$

such that

- Π-ren is a renamed version of the state template Π where each operator f in Π has been replaced by a fresh new operator $f^{\triangleright}$;
- `fail` is a fresh new constant that does not occur in the signature of $\mathcal{R}$;
- $ori(t^{\triangleright})$ is a function that takes a renamed term $t^{\triangleright}$ and returns its original version t, that is, $ori(t^{\triangleright}) = t$.

Note that assertion-checking is executed over renamed versions of the original program states. When a (subterm of a) renamed state matches Π-ren, via a substitution $\sigma^{\triangleright}$, and the associated instance of the state invariant in the original term algebra $ori(\varphi\sigma^{\triangleright})$ does not hold, a `fail` term is yielded and consequently an assertion violation is signaled. Renaming is key in order to neatly separate assertion checking from system computations so that they do not interfere: the overall system execution uses the original operators and equations in the equational theory, while assertion violation detection is performed on renamed terms (for a detailed discussion on renaming, see [4]).

Example 4. *Given the safety policy* $\mathcal{A}_{\mathtt{DAM}}$ *of Example* 3 *and the Maude program* $\mathcal{R}_{\mathtt{DAM}} = (\Sigma_D, E_D, R_D)$ *of Example* 1*, the set* $Eq(\mathcal{A}_{\mathtt{DAM}})$ *includes the following equations*[2]*:*

```
(e1) { SC | V | T | AC }▷ = fail if  not(ori((V < 50000000))) .


(e2) { [ S1,O1 ]▷ [ S2,O2 ]▷ [ S3,O3 ]▷  | V:Rat | T:TimeStamp | AC:Bool }▷ = fail
                    if not(ori((V:Rat > 40000000) implies (
```

[2] Note that, in the case of mixfix operators, we just rename one operator symbol. For instance, the constructor operator for system states $\{\_|\_|\_|\_\}$ is renamed $\{\_|\_|\_|\_\}^{\triangleright}$.

```
                           (O1 == open3 and O2 =/= close and O3 =/= close) or
                           (O2 == open3 and O1 =/= close and O3 =/= close) or
                           (O3 == open3 and O1 =/= close and O2 =/= close))
                         )) .

(e3) { SC | V | T | AC }▷ = fail if not(ori((V < 10000000) implies
                                              (openSpillways(SC) == 0)
                                             )) .

(e4) { SC | V | T | AC }▷ = fail if not(ori(((V >= 10000000) and (V <= 40000000))
                                             implies
                                              (openSpillways(SC) == 2)
                                             )) .
```

Note that the renamed state

$\{\texttt{[s1}^{\rhd}\texttt{,open3}^{\rhd}\texttt{]}^{\rhd}\ \texttt{[s2}^{\rhd}\texttt{,open1}^{\rhd}\texttt{]}^{\rhd}\ \texttt{[s3}^{\rhd}\texttt{,open1}^{\rhd}\texttt{]}^{\rhd}\ \texttt{|}\ \texttt{50100000}^{\rhd}\ \texttt{|}\ \texttt{20}^{\rhd}\ \texttt{|}\ \texttt{true}^{\rhd}\}^{\rhd}$

can be reduced to `fail` by using the extended equational theory $E_D \cup Eq(\mathcal{A}_{\texttt{DAM}})$ (specifically via the application of Equation `e1`).

Phase 2. Transforming the rewrite rules into guarded rewrite rules. This phase transforms the original rewrite rules of $\mathcal{R}$ into guarded, conditional rewrite rules that can only be fired if no system assertion is violated. Intuitively, this is achieved by transforming each rewrite rule $r : (\lambda \Rightarrow\ \rho\ \texttt{if C})$ of $\mathcal{R}$ into a specialized version $r^{\rhd}$: $(\lambda \Rightarrow\ \rho\ \texttt{if C} \wedge \texttt{check}(\rho)\ \texttt{=/=}\ \texttt{fail})$ of r that contains the extra constraint $\texttt{check}(\rho)\ \texttt{=/=}\ \texttt{fail}$ that holds when the *renamed* instances of the right-hand side ρ cannot by simplified to `fail` by using the extended equational theory $E \cup Eq(\mathcal{A})$. This way, we ensure that any state transition $t_1 \xrightarrow{r^{\rhd}}_{E \cup Eq(\mathcal{A})} t_2$, which yields the system state t_2 by means of the application of the rule $r^{\rhd}$, is enabled only if t_2 is a safe state w.r.t. the assertions in $\mathcal{A}$.

Example 5. *Given the safety policy $\mathcal{A}_{\texttt{DAM}}$ of Example 3 and the Maude program $\mathcal{R}_{\texttt{DAM}} = (\Sigma_D, E_D, R_D)$ of Example 1, the rewrite rules in R_D are specialized as follows:*

```
rl [nocmd] : { SC | V | T | true } => { SC | V | T | false }
                                       if check({ SC | V | T | false }) == fail .
rl [openC-1] :
    { [S,close] SS | V | T | true } => { [S,open1] SS | V | T | false }
                             if check({ [S,open1] SS | V | T | false }) == fail .
rl [open1-2] :
    { [S,open1] SS | V | T | true } => { [S,open2] SS | V | T | false }
                             if check({ [S,open3] SS | V | T | false }) == fail .
rl [open2-3] :
    { [S,open2] SS | V | T | true } => { [S,open3] SS | V | T | false }
                             if check({ [S,open3] SS | V | T | false }) == fail .
rl [close1-C] :
    { [S,open1] SS | V | T | true } => { [S,close] SS | V | T | false }
                             if check({ [S,close] SS | V | T | true }) == fail .
rl [close2-1] :
    { [S,open2] SS | V | T | true } => { [S,open1] SS | V | T | false }
                             if check({ [S,open1] SS | V | T | false }) == fail .
rl [close3-2] :
    { [S,open3] SS | V | T | true } => { [S,open2] SS | V | T | false }
                             if check({ [S,open2] SS | V | T | false }) == fail .
crl [volume] : { SC | V | T | false } => { SC | V' | (T + deltaT) | true }
                       if V' := (V + inflow * deltaT) - (outflow(SC) * deltaT) /\
                          check({ SC | V' | (T + deltaT) | true }) == fail .
```

As formally proven in [4], computations in the resulting program $\mathcal{R}^{\rhd}$ are both reproducible in $\mathcal{R}$ and guaranteed to meet $\mathcal{A}$. In other words, for each computation $\mathcal{C}$ in $\mathcal{R}^{\rhd}$, (i) $\mathcal{C}$ is also a computation in $\mathcal{R}$, and (ii) there is no system state t in $\mathcal{C}$ that violates one or more system assertions of $\mathcal{A}$.

ÁTAME has been coded in Maude itself by using Maude's meta-level capabilities. It integrates a RESTful Web service that is written in Java and an intuitive Web user interface that is based on AJAX technology and is written in HTML5 and Javascript. The implementation contains about 600 lines of Maude source code, 600 lines of C++ code, 750 lines of Java code, and 700 lines of HTML5 and JavaScript code. The tool is publicly available together with a number of examples at `http://safe-tools.dsic.upv.es/atame`.

4.1. ÁTAME Additional Features

In addition to program specialization w.r.t. logical assertions, we endowed ÁTAME with additional features that allow the runtime behaviour of the computed safe program specializations to be analyzed and improved. More specifically, the ÁTAME tool supports the incremental exploration of the computation space for the specialized program as well as the selection of specific computations within it.

Incremental Exploration. Roughly speaking, given a system state s, the computation tree that is rooted at s can be stepwisely inspected by expanding any frontier state s_i in the currently deployed tree fragment (stemming from s) with all rewrite steps $s_i \longrightarrow_E s'_i$. Thanks to this feature, ÁTAME provides a bird's-eye view over the computation space of the computed specialization $\mathcal{R}^{\triangleright}$ that can be inspected to check whether the desired runtime behaviour of $\mathcal{R}^{\triangleright}$ was achieved or the considered safety policy $\mathcal{A}$ needs to be recalibrated in order to discharge a stronger/weaker specialization (which can be obtained by reapplying the transformation). This facility is implemented by interconnecting ÁTAME with ANIMA [6] —a program animator for Maude programs that allows programs to be executed by incrementally building and visualizing their computation trees. Moreover, it also supports a graph representation of the state space that can improve the user's understanding of the program behavior.

Computation picking. ÁTAME allows searching for a (deterministic) computation inside the computation space of the transformed program. The search is modeled as a reachability problem

$$(s_i \Rightarrow^* S_f \textit{ such that } C)$$

where s_i is a system state, S_f is a state template, and C is a Boolean condition that is used to filter out undesired solutions. Solving $(s_i \Rightarrow^* S_f \textit{ such that } C)$ essentially amounts to generating a computation that connects s_i with a final state s_f such that s_f matches the state structure S_f (modulo Ax) with a substitution σ and $C\sigma$ holds. If such a computation does not exist, the reachability problem has no solution. This feature plays a fundamental role in a scenario in which the Maude program models a nondeterministic system controller and a distinguished computation must be extracted from the system computation space, which is done by identifying a deterministic sequence of actions that allow a critical configuration to transition to a safe configuration.

Example 6. *As an illustrative example, consider the Maude program $\mathcal{R}_{\texttt{DAM}}$ and its safe specialization $\mathcal{R}^{\triangleright}_{\texttt{DAM}}$ that meets the safety policy $\mathcal{A}_{\texttt{DAM}}$ of Example* 5*. Let s_i be the initial state*

```
{ [s1,open3] [s2,open3] [s3,open2] | 49970000 | 0 | true }
```

that describes a dam configuration at time 0 *where the basin water volume is particularly high. Then, by feeding ÁTAME with the following reachability problem*

```
{ [s1,open3] [s2,open3] [s3,open2] | 49970000 | 0 | true } ⇒*
{ [s1,open3] [s2,open3] [s3,open3] | V | T | true ] }
                                  such that V < 49960000 and T < 60
```

a computation $\mathcal{C}_{safe}$ in $\mathcal{R}^{\triangleright}_{\texttt{DAM}}$ is automatically delivered that represents the sequence of actions that must be taken to bring the basin water volume from $49970000m^3$ *to a safer volume below*

$49960000m^3$ in less than 60 time units. Furthermore, note that the computation $\mathcal{C}_{safe}$ is safe, since it has been searched in the safe-by-construction specialized program $\mathcal{R}^{\triangleright}_{\mathtt{DAM}}$. Hence, each intermediate state in $\mathcal{C}_{safe}$ does not break any assertion in $\mathcal{A}_{\mathtt{DAM}}$.

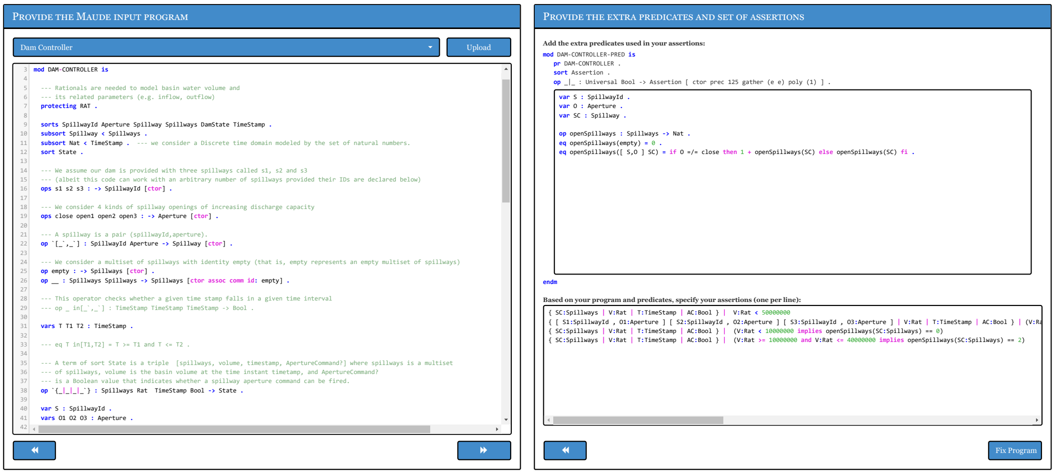
Computation picking has been implemented by making use of the (meta-)search Maude capability that solves reachability problems by visiting the program computation tree in a breadth-first fashion. Since computation trees may include nonterminating computations, a depth bound must be set to limit the search within the tree and make the search feasible. A search with depth d analyzes the finite computation tree fragment that consists of all the computations with at most d rewrite rule applications.

Also, ÁTAME is interconnected with iJulienne [10], a trace slicer for Maude programs that allows the delivered computation to be graphically visualized and simplified by applying a slicing technique that removes undesired data (i.e., data the user does not want to observe) from the trace. More precisely, the user first selects a set of symbols to be observed from the last state of the computation (i.e., a slicing criterion); then, trace slicing automatically deletes from the computation all the data that are irrelevant to the criterion of interest (i.e., that are not needed to produce the observed data). For a detailed discussion on trace slicing, please refer to [1].

5. ÁTAME in Action

This section shows how ÁTAME works in practice by describing a typical program specialization session in which the dam controller $\mathcal{R}_{\mathtt{DAM}}$ is specialized w.r.t. the safety policy $\mathcal{A}_{\mathtt{DAM}}$.

Maude programs can be uploaded in ÁTAME as simple `.maude` module files, written from scratch inside a dedicated edit area, or selected from a preloaded collection of Maude programs that is provided with the tool for demonstration purposes. In this case, to start the program specialization session, we select the `DAM-CONTROLLER` Maude module from the preloaded programs (see Figure 4 (Left)), which models the dam controller $\mathcal{R}_{\mathtt{DAM}}$ of Example 1. Next, we input the system assertions that specify the safety policy $\mathcal{A}_{\mathtt{DAM}}$ of Example 3 together with the additional function `openSpillways`, which is used in the formalization of $\mathcal{A}_{\mathtt{DAM}}$ itself (see Figure 4 (Right)).



```
mod DAM-CONTROLLER is

  --- Rationals are needed to model basin water volume and
  --- its related parameters (e.g. inflow, outflow)
  protecting RAT .

  sorts SpillwayId Aperture Spillway Spillways DamState TimeStamp .
  subsort Spillway < Spillways .
  subsort Nat < TimeStamp .  --- we consider a Discrete time domain modeled by the set of natural numbers.
  sort State .

  --- We assume our dam is provided with three spillways called s1, s2 and s3
  --- (albeit this code can work with an arbitrary number of spillways provided their IDs are declared below)
  ops s1 s2 s3 : -> SpillwayId [ctor] .

  --- We consider 4 kinds of spillway openings of increasing discharge capacity
  ops close open1 open2 open3 : -> Aperture [ctor] .

  --- A spillway is a pair (spillwayId,aperture).
  op `[_`,_`] : SpillwayId Aperture -> Spillway [ctor] .

  --- We consider a multiset of spillways with identity empty (that is, empty represents an empty multiset of spillways)
  op empty : -> Spillways [ctor] .
  op __ : Spillways Spillways -> Spillways [ctor assoc comm id: empty] .

  --- This operator checks whether a given time stamp falls in a given time interval
  --- op _ in[_`,_`] : TimeStamp TimeStamp TimeStamp -> Bool .

  vars T T1 T2 : TimeStamp .

  --- eq T in[T1,T2] = T >= T1 and T <= T2 .

  --- A term of sort State is a triple  [spillways, volume, timestamp, ApertureCommand?] where spillways is a multiset
  --- of spillways, volume is the basin volume at the time instant timetamp, and ApertureCommand?
  --- is a Boolean value that indicates whether a spillway aperture command can be fired.
  op `{_|_|_|_`} : Spillways Rat  TimeStamp Bool -> State .

  var S : SpillwayId .
  vars O1 O2 O3 : Aperture .
```

PROVIDE THE EXTRA PREDICATES AND SET OF ASSERTIONS
Add the extra predicates used in your assertions:

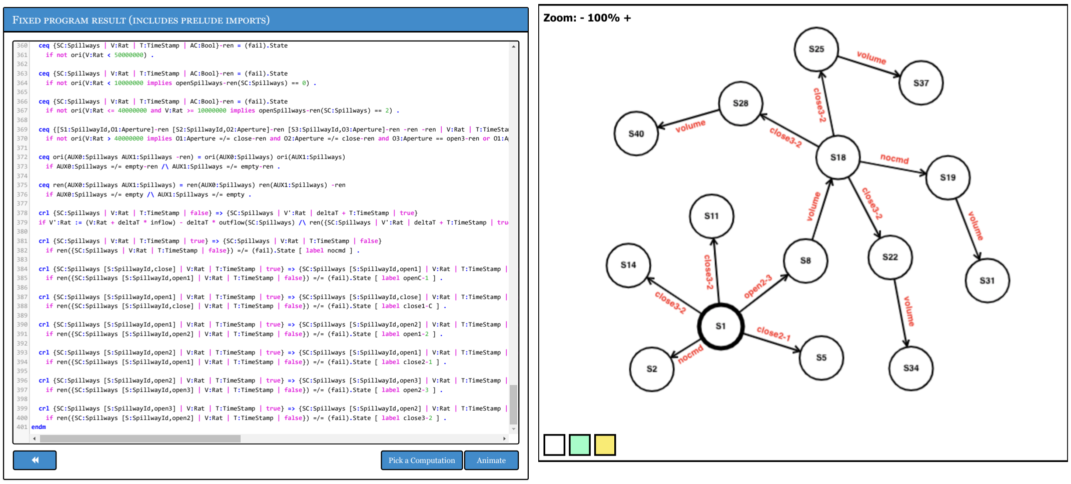
```
mod DAM-CONTROLLER-PRED is
   pr DAM-CONTROLLER .
   sort Assertion .
   op _|_ : Universal Bool -> Assertion [ ctor prec 125 gather (e e) poly (1) ] .
    var S : SpillwayId .
    var O : Aperture .
    var SC : Spillway .

    op openSpillways : Spillways -> Nat .
    eq openSpillways(empty) = 0 .
    eq openSpillways([ S,O ] SC) = if O =/= close then 1 + openSpillways(SC) else openSpillways(SC) fi .
endm
```

Based on your program and predicates, specify your assertions (one per line):

```
{ SC:Spillways | V:Rat | T:TimeStamp | AC:Bool } |  V:Rat < 50000000
{ [ S1:SpillwayId , O1:Aperture ] [ S2:SpillwayId , O2:Aperture ] [ S3:SpillwayId , O3:Aperture ] | V:Rat | T:TimeStamp | AC:Bool } | (V:Ra
{ SC:Spillways | V:Rat | T:TimeStamp | AC:Bool } |  (V:Rat < 10000000 implies openSpillways(SC:Spillways) == 0)
{ SC:Spillways | V:Rat | T:TimeStamp | AC:Bool } |  (V:Rat >= 10000000 and V:Rat <= 40000000 implies openSpillways(SC:Spillways) == 2)
```

FIGURE 4. (Left) Loading the `DAM-CONTROLLER` Maude module in ÁTAME. (Right) Loading the safety policy $\mathcal{A}_{\mathtt{DAM}}$ in ÁTAME.

At this point, by pressing the FIX PROGRAM button, ÁTAME automatically generates the program specialization $\mathcal{R}^{\triangleright}_{\mathtt{DAM}}$ of $\mathcal{R}_{\mathtt{DAM}}$, which is safe w.r.t. $\mathcal{A}_{\mathtt{DAM}}$. Figure 5 (Left) shows a fragment of such a specialization that includes $Eq(\mathcal{A}_{\mathtt{DAM}})$ (i.e., the equations for detecting assertion violations described in Example 4) and the specialized rewrite rules of Example 5.



```
ceq {SC:Spillways | V:Rat | T:TimeStamp | AC:Bool}-ren = (fail).State
  if not ori(V:Rat < 50000000) .

ceq {SC:Spillways | V:Rat | T:TimeStamp | AC:Bool}-ren = (fail).State
  if not ori(V:Rat < 10000000 implies openSpillways-ren(SC:Spillways) == 0) .

ceq {SC:Spillways | V:Rat | T:TimeStamp | AC:Bool}-ren = (fail).State
  if not ori(V:Rat <= 40000000 and V:Rat >= 10000000 implies openSpillways-ren(SC:Spillways) == 2) .

ceq {[S1:SpillwayId,O1:Aperture]-ren [S2:SpillwayId,O2:Aperture]-ren [S3:SpillwayId,O3:Aperture]-ren -ren -ren | V:Rat | T:TimeSta
  if not ori(V:Rat > 40000000 implies O1:Aperture =/= close-ren and O2:Aperture =/= close-ren and O3:Aperture == open3-ren or O1:A

ceq ori(AUX0:Spillways AUX1:Spillways -ren) = ori(AUX0:Spillways) ori(AUX1:Spillways)
  if AUX0:Spillways =/= empty-ren /\ AUX1:Spillways =/= empty-ren .

ceq ren(AUX0:Spillways AUX1:Spillways) = ren(AUX0:Spillways) ren(AUX1:Spillways) -ren
  if AUX0:Spillways =/= empty /\ AUX1:Spillways =/= empty .

crl {SC:Spillways | V:Rat | T:TimeStamp | false} => {SC:Spillways | V':Rat | deltaT + T:TimeStamp | true}
if V':Rat := (V:Rat + deltaT * inflow) - deltaT * outflow(SC:Spillways) /\ ren({SC:Spillways | V':Rat | deltaT + T:TimeStamp | tru

crl {SC:Spillways | V:Rat | T:TimeStamp | true} => {SC:Spillways | V:Rat | T:TimeStamp | false}
  if ren({SC:Spillways | V:Rat | T:TimeStamp | false}) =/= (fail).State [ label nocmd ] .

crl {SC:Spillways [S:SpillwayId,close] | V:Rat | T:TimeStamp | true} => {SC:Spillways [S:SpillwayId,open1] | V:Rat | T:TimeStamp |
  if ren({SC:Spillways [S:SpillwayId,open1] | V:Rat | T:TimeStamp | false}) =/= (fail).State [ label openC-1 ] .

crl {SC:Spillways [S:SpillwayId,open1] | V:Rat | T:TimeStamp | true} => {SC:Spillways [S:SpillwayId,close] | V:Rat | T:TimeStamp |
  if ren({SC:Spillways [S:SpillwayId,close] | V:Rat | T:TimeStamp | false}) =/= (fail).State [ label close1-C ] .

crl {SC:Spillways [S:SpillwayId,open1] | V:Rat | T:TimeStamp | true} => {SC:Spillways [S:SpillwayId,open2] | V:Rat | T:TimeStamp |
  if ren({SC:Spillways [S:SpillwayId,open2] | V:Rat | T:TimeStamp | false}) =/= (fail).State [ label open1-2 ] .

crl {SC:Spillways [S:SpillwayId,open2] | V:Rat | T:TimeStamp | true} => {SC:Spillways [S:SpillwayId,open1] | V:Rat | T:TimeStamp |
  if ren({SC:Spillways [S:SpillwayId,open1] | V:Rat | T:TimeStamp | false}) =/= (fail).State [ label close2-1 ] .

crl {SC:Spillways [S:SpillwayId,open2] | V:Rat | T:TimeStamp | true} => {SC:Spillways [S:SpillwayId,open3] | V:Rat | T:TimeStamp |
  if ren({SC:Spillways [S:SpillwayId,open3] | V:Rat | T:TimeStamp | false}) =/= (fail).State [ label open2-3 ] .

crl {SC:Spillways [S:SpillwayId,open3] | V:Rat | T:TimeStamp | true} => {SC:Spillways [S:SpillwayId,open2] | V:Rat | T:TimeStamp |
  if ren({SC:Spillways [S:SpillwayId,open2] | V:Rat | T:TimeStamp | false}) =/= (fail).State [ label close3-2 ] .
endm
```

FIGURE 5. (Left) A fragment of the safe specialization for $\mathcal{R}_{\mathtt{DAM}}$ computed by ÁTAME. (Right) A graph representation of the computation tree in Figure 6.

Once the specialization $\mathcal{R}^{\triangleright}_{\mathtt{DAM}}$ has been computed, the user is given the possibility to analyze it in two ways. First, by pressing the ANIMATE button, $\mathcal{R}^{\triangleright}_{\mathtt{DAM}}$ is executed by incrementally building and exploring the computation tree of $\mathcal{R}^{\triangleright}_{\mathtt{DAM}}$ w.r.t. a given input initial state. For instance, in Figure 6, we show a fragment of the computation tree of $\mathcal{R}^{\triangleright}_{\mathtt{DAM}}$ w.r.t. the initial state

$$s = \{\ \texttt{[s1,open3] [s2,open3] [s3,open3] | 49990000 | 0 | true }\}.$$

Note that all of the states in the considered tree fragment fulfill the system assertions formalized in $\mathcal{A}_{\mathtt{DAM}}$. Specifically, it can be observed that the basin water volume is below the established critical threshold (50 millions m^3) in every system state (as specified by Assertion a1). Furthermore, in each state, all of the spillways are open and at least one has a maximum aperture width (open3) as required by Assertion a2 whenever the water volume is greater than 40 million m^3. A more compact, graph representation of the deployed computation tree can be visualized by selecting the

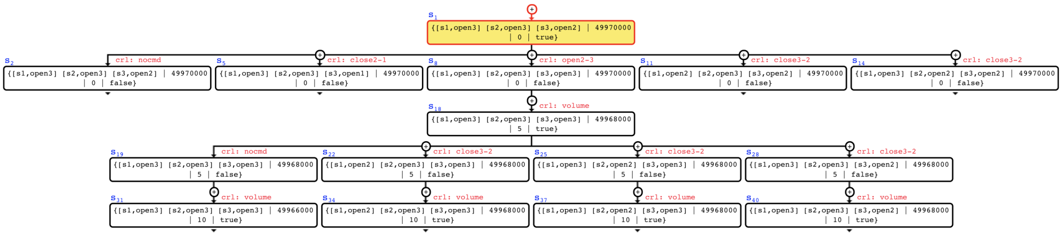
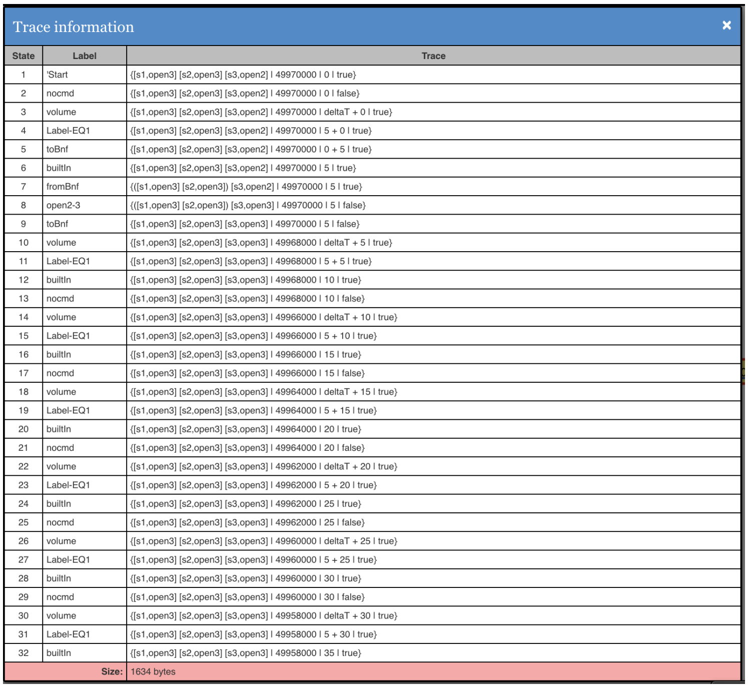

s1
{[s1,open3] [s2,open3] [s3,open2] | 49970000 | 0 | true}
crl: nocmd
s2
{[s1,open3] [s2,open3] [s3,open2] | 49970000 | 0 | false}
crl: close2-1
s5
{[s1,open3] [s2,open3] [s3,open1] | 49970000 | 0 | false}
crl: open2-3
s8
{[s1,open3] [s2,open3] [s3,open3] | 49970000 | 0 | false}
crl: close3-2
s11
{[s1,open2] [s2,open3] [s3,open2] | 49970000 | 0 | false}
crl: close3-2
s14
{[s1,open3] [s2,open2] [s3,open2] | 49970000 | 0 | false}
crl: volume
s18
{[s1,open3] [s2,open3] [s3,open3] | 49968000 | 5 | true}
crl: nocmd
s19
{[s1,open3] [s2,open3] [s3,open3] | 49968000 | 5 | false}
crl: close3-2
s22
{[s1,open2] [s2,open3] [s3,open3] | 49968000 | 5 | false}
crl: close3-2
s25
{[s1,open3] [s2,open2] [s3,open3] | 49968000 | 5 | false}
crl: close3-2
s28
{[s1,open3] [s2,open3] [s3,open2] | 49968000 | 5 | false}
crl: volume
s31
{[s1,open3] [s2,open3] [s3,open3] | 49966000 | 10 | true}
crl: volume
s34
{[s1,open2] [s2,open3] [s3,open3] | 49968000 | 10 | true}
crl: volume
s37
{[s1,open3] [s2,open2] [s3,open3] | 49968000 | 10 | true}
crl: volume
s40
{[s1,open3] [s2,open3] [s3,open2] | 49968000 | 10 | true}

FIGURE 6. A computation tree fragment of $\mathcal{R}^{\triangleright}_{\mathtt{DAM}}$ w.r.t. the initial state {[s1,open3] [s2,open3] [s3,open2] | 49970000 | 0 | true}.

Draw computation graph option in the settings menu, which is accessible through the wheel icon (see Figure 5 (Right)).

Second, by pressing the PICK A COMPUTATION button, the system asks to input a reachability problem $(s_i \Rightarrow^* S_f \text{ where } Cond)$ plus an integer that indicates the maximum search depth within the computation tree. Then, a search starts and a suitable deterministic computation is singled out (if any). Thanks to the interconnection with the iJulienne trace slicer, the produced computation can be automatically visualized in a tabular form or as a browsable sequence of state transitions that the user can navigate through and eventually simplify by means of IJULIENNE trace slicing capabilities. For instance, the reachability problem of Example 6 yields the safe computation $\mathcal{C}_{safe}$ in Figure 7 when the associated computation tree is searched with a search depth equal to 20. $\mathcal{C}_{safe}$ consists of 32 systems states (including both rule applications and equational and axiomatic simplifications) and pinpoints a deterministic sequence of actions that brings the initial critical state { `[s1,open3] [s2,open3] [s3,open2] | 49970000 | 0 | true` } to the safe state { `[s1,open3] [s2,open3] [s3,open3] | 49958000 | 35 | true` } in exactly 35 units of time.



Trace information

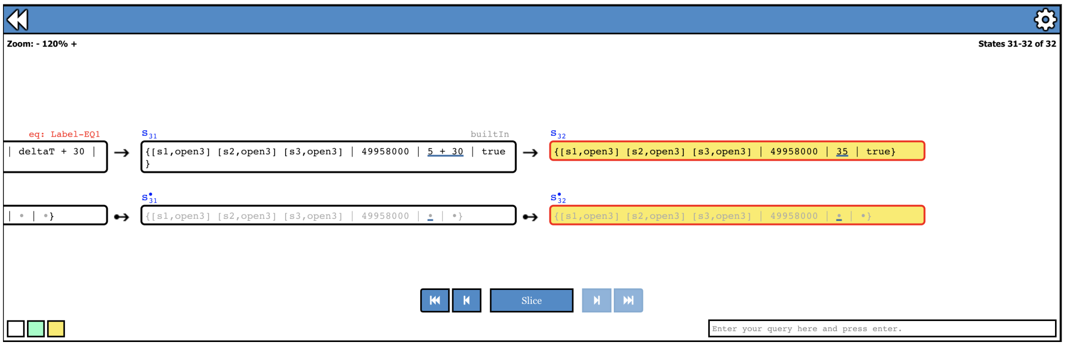
State	Label	Trace
1	'Start	{[s1,open3] [s2,open3] [s3,open2] | 49970000 | 0 | true}
2	nocmd	{[s1,open3] [s2,open3] [s3,open2] | 49970000 | 0 | false}
3	volume	{[s1,open3] [s2,open3] [s3,open2] | 49970000 | deltaT + 0 | true}
4	Label-EQ1	{[s1,open3] [s2,open3] [s3,open2] | 49970000 | 5 + 0 | true}
5	toBnf	{[s1,open3] [s2,open3] [s3,open2] | 49970000 | 0 + 5 | true}
6	builtIn	{[s1,open3] [s2,open3] [s3,open2] | 49970000 | 5 | true}
7	fromBnf	{([s1,open3] [s2,open3]) [s3,open2] | 49970000 | 5 | true}
8	open2-3	{([s1,open3] [s2,open3]) [s3,open3] | 49970000 | 5 | false}
9	toBnf	{[s1,open3] [s2,open3] [s3,open3] | 49970000 | 5 | false}
10	volume	{[s1,open3] [s2,open3] [s3,open3] | 49968000 | deltaT + 5 | true}
11	Label-EQ1	{[s1,open3] [s2,open3] [s3,open3] | 49968000 | 5 + 5 | true}
12	builtIn	{[s1,open3] [s2,open3] [s3,open3] | 49968000 | 10 | true}
13	nocmd	{[s1,open3] [s2,open3] [s3,open3] | 49968000 | 10 | false}
14	volume	{[s1,open3] [s2,open3] [s3,open3] | 49966000 | deltaT + 10 | true}
15	Label-EQ1	{[s1,open3] [s2,open3] [s3,open3] | 49966000 | 5 + 10 | true}
16	builtIn	{[s1,open3] [s2,open3] [s3,open3] | 49966000 | 15 | true}
17	nocmd	{[s1,open3] [s2,open3] [s3,open3] | 49966000 | 15 | false}
18	volume	{[s1,open3] [s2,open3] [s3,open3] | 49964000 | deltaT + 15 | true}
19	Label-EQ1	{[s1,open3] [s2,open3] [s3,open3] | 49964000 | 5 + 15 | true}
20	builtIn	{[s1,open3] [s2,open3] [s3,open3] | 49964000 | 20 | true}
21	nocmd	{[s1,open3] [s2,open3] [s3,open3] | 49964000 | 20 | false}
22	volume	{[s1,open3] [s2,open3] [s3,open3] | 49962000 | deltaT + 20 | true}
23	Label-EQ1	{[s1,open3] [s2,open3] [s3,open3] | 49962000 | 5 + 20 | true}
24	builtIn	{[s1,open3] [s2,open3] [s3,open3] | 49962000 | 25 | true}
25	nocmd	{[s1,open3] [s2,open3] [s3,open3] | 49962000 | 25 | false}
26	volume	{[s1,open3] [s2,open3] [s3,open3] | 49960000 | deltaT + 25 | true}
27	Label-EQ1	{[s1,open3] [s2,open3] [s3,open3] | 49960000 | 5 + 25 | true}
28	builtIn	{[s1,open3] [s2,open3] [s3,open3] | 49960000 | 30 | true}
29	nocmd	{[s1,open3] [s2,open3] [s3,open3] | 49960000 | 30 | false}
30	volume	{[s1,open3] [s2,open3] [s3,open3] | 49958000 | deltaT + 30 | true}
31	Label-EQ1	{[s1,open3] [s2,open3] [s3,open3] | 49958000 | 5 + 30 | true}
32	builtIn	{[s1,open3] [s2,open3] [s3,open3] | 49958000 | 35 | true}
	Size:	1634 bytes

FIGURE 7. The computation $\mathcal{C}_{safe}$ delivered by ÁTAME as the outcome of the reachability problem of Example 6.

Finally, the browsable version of $\mathcal{C}_{safe}$ is shown in Figure 8 together with a companion computation slice that has been obtained from $\mathcal{C}_{safe}$ by choosing a slicing criterion that ignores the `timestamp` and `apertureCommand` flag.

6. Experimental evaluation

We benchmarked the performance of the ATAME system against the following collection of Maude programs, which are all available and fully described within the ATAME Web platform: *Bank model*,



```
| deltaT + 30 |  →  {[s1,open3] [s2,open3] [s3,open3] | 49958000 | 5 + 30 | true
                    }                                                          →  {[s1,open3] [s2,open3] [s3,open3] | 49958000 | 35 | true}

| • | •}        •→  {[s1,open3] [s2,open3] [s3,open3] | 49958000 | • | •}     •→  {[s1,open3] [s2,open3] [s3,open3] | 49958000 | • | •}
```

FIGURE 8. A browsable version of $\mathcal{C}_{safe}$ and its sliced counterpart.

a conditional Maude specification that models a distributed banking system; *Blocks World*, a Maude encoding of the classical AI planning problem that consists of setting one or more vertical stacks of blocks on a table using a robotic arm; *BRP*, a Maude implementation of the Bounded Retransmission Protocol; *Container*, a Maude specification that models the cargo manipulation in a container terminal; *DAM controller*, our running example; *Dekker*, a Maude specification of Dekker's mutual exclusion algorithm; *Maude NPA*, an analysis tool for cryptographic protocols that takes into account the algebraic properties of cryptosystems; *Philosophers*, a Maude specification of Dijkstra's classical concurrency example; *Semaphore*, a classical mutual exclusion protocol with semaphores written in Maude; *Stock Exchange*, a simplified stock exchange concurrent system in which traders operate on stocks via limit orders; *Webmail app*, a Maude specification of a rich webmail application that provides typical email management, system administration capabilities, login/logout functionality, etc; *Wolfram's Rule 30*, a one-dimensional binary cellular automaton rule introduced by Stephen Wolfram.

To conduct our experiments, we have used the following *modus operandi*.

1. Each Maude program $\mathcal{R}$ in the benchmark set is specialized w.r.t. three safety policies with an increasing number of system assertions (10, 25, 50 assertions); the generation time $G_{\mathcal{R}^{\triangleright}}$ for building each safe program specialization $\mathcal{R}^{\triangleright}$ is recorded. Furthermore, the number of equations[3] in $\mathcal{R}$ and $\mathcal{R}^{\triangleright}$ is used as a measure of their size (columns $\#\mathcal{E}_{\mathcal{R}}$ and $\#\mathcal{E}_{\mathcal{R}^{\triangleright}}$, respectively).
2. The execution times of the original program ($T_{\mathcal{R}}$) and of each program specialization ($T_{\mathcal{R}^{\triangleright}}$) are measured w.r.t. computations that consist of about 500 Maude computation steps, which amounts to 5,000 rewrite steps on average including equational simplification steps. Times are measured in milliseconds. Each measure has been repeated 10 times and the average has been taken to reduce the computation noise.

The experiments have been performed on a machine equipped with a 3.3GHz Intel Xeon E5-1660 CPU and 64GB RAM. The results are shown in Table 1. Our figures show that, on average, the increasing in the number of equations grows linearly to the number of assertions and the size of the corrected code is 11.8 times the size of the original code. For instance, for the bank model, the increase in size ranges from 7.1 for 10 assertions to 8.1 for 50 assertions. Nonetheless, column $G_{\mathcal{R}^{\triangleright}}$ shows rather good generation times (a few milliseconds) in each case. On average, generating a new equation of $\mathcal{R}^{\triangleright}$ takes less then 1ms. Generation times also scale well w.r.t. the size (column $\#\mathcal{A}$) of each safety policy under examination. This demonstrates that our approach works well in practice even when large assertion sets are considered.

[3] We do not include rewrite rules in the computation of the program size of $\mathcal{R}$ and $\mathcal{R}^{\triangleright}$, since both programs have the same number of rewrite rules.

	$T_{\mathcal{R}}$	$\#\mathcal{E}_{\mathcal{R}}$	$\#\mathcal{A}$	$\#\mathcal{E}_{\mathcal{R}^{\triangleright}}$	$G_{\mathcal{R}^{\triangleright}}$	$T_{\mathcal{R}^{\triangleright}}$	$O_{\mathcal{R}^{\triangleright}}$
Bank model	17	38	10	271	15	37	1.18
			25	286	16	42	1.47
			50	311	24	51	2.00
Blocks world	19	12	10	211	9	40	1.11
			25	226	15	48	1.53
			50	251	25	55	1.89
BRP	5	12	10	87	8	8	0.60
			25	102	13	9	0.80
			50	127	24	10	1.00
Container	14	20	10	226	16	20	0.43
			25	241	26	21	0.50
			50	266	41	25	0.79
Dam Controller	11	66	10	329	21	23	1.08
			25	344	24	24	1.18
			50	369	39	26	1.36
Dekker	40	25	10	341	23	52	0.30
			25	356	25	53	0.33
			50	381	36	55	0.38
Maude-NPA	33	11	10	179	6	40	0.21
			25	194	9	42	0.27
			50	219	19	44	0.33
Philosophers	12	12	10	144	8	15	0.25
			25	159	11	15	0.25
			50	184	25	16	0.33
Semaphore Problem	7	10	10	131	9	17	1.43
			25	146	12	22	2.14
			50	171	28	24	2.43
Stock Ex. model	36	106	10	419	33	56	0.56
			25	434	62	74	1.06
			50	459	81	85	1.36
Webmail app	138	191	10	1167	113	194	0.41
			25	1182	128	198	0.43
			50	1207	138	201	0.46
Wolfram's Rule 30	7	13	10	138	9	10	0.43
			25	153	16	12	0.71
			50	178	25	15	1.14

TABLE 1. Experimental results of the specialization technique.

As for the execution times $T_{\mathcal{R}}$ and $T_{\mathcal{R}^{\triangleright}}$, our figures show that $T_{\mathcal{R}^{\triangleright}}$ is slightly greater than $T_{\mathcal{R}}$ for every experiment. This slowdown is obviously expected since $\mathcal{R}^{\triangleright}$ comes with a more complex equational structure that is used to perform the safety checks that are encoded in the conditional rewrite rules of $\mathcal{R}^{\triangleright}$. However, even when the size of $\mathcal{R}^{\triangleright}$ is much bigger than the size of $\mathcal{R}$, the running times scale linearly with the number of assertions. Actually, our figures reveal reasonable overheads (Column $O_{\mathcal{R}^{\triangleright}}$), which we compute as $(T_{\mathcal{R}^{\triangleright}} - T_{\mathcal{R}})/T_{\mathcal{R}}$, with the average overhead being less than one (0.89). And moreover, the incurred cost is negligible when compared to the high execution costs of monitored runtime environments, which can be to five times the running time of the original program (e.g., for the Bank Model with only five assertions, see [4]). Overall, the correction transformation is useful since the transformed program is demonstrably safe and corrections are generated fast enough that they could be computed during active development, thus reducing the debugging burden.

7. Concluding Remarks

The technique described in this paper presents similarities with automated program correction and related problems such as code fixing and repair techniques. The discussion of these similarities is outside the scope of this paper; a detailed comparison can be found in [4].

In order to guarantee that a program satisfies a property, three kinds of approaches have been considered in the literature: static, dynamic, and mixed approaches [8]. Static approaches such as program derivation, static analysis, and type-checking can ensure properties without any runtime penalty. However they are generally dedicated to limited properties and may reject perfectly correct programs. Dynamic approaches such as code instrumentation and runtime monitors rely on some kind of state machine that closely follows the program execution and halts the program whenever it is about to violate the property. The main drawback of dynamic approaches is their runtime costs. Policy weaving [11] is a mixed program transformation technique that rewrites a program so that it is guaranteed to be safe with respect to a security policy. Its promise stems from the possibility of blending the best aspects of static and dynamic analysis as it relies on (i) static analysis to identify program points at which policy violations might occur, and (ii) runtime analysis to implement dynamic enforcement. Policy weaving is often combined with transactional introspection, which allows the state resulting from the execution of a statement to be examined and, if the policy is about to be violated, the state is suppressed. In this paper, we advocate a different approach that also avoids any undesirable states to be produced but that is essentially static in the sense that it applies a (source-to-source) program transformation that specializes the program code to the properties of interest. Nevertheless, our approach could also be classified as a dynamic, programming language approach according to [8] in the sense that it integrates the runtime checks within the program itself yet achieves them with no prohibitive runtime overhead. Moreover, this technique does not reject correct programs and supports the efficient enforcement of strong, customized, invariant properties that are specified in a purely declarative way.

Liquid Haskell (LH) [14] allows Haskell code to be annotated with data type invariants that complement the invariants imposed by the types with logical predicates; this allows safety properties to be enforced at compile time. A liquid type has the form $\{v : \tau | e\}$, where τ is a Hindley-Milner type and e is a boolean expression and represents all the values u of type τ such that the expression $e[u/v]$ evaluates to true. Liquid type annotations are provided by the programmer in the input file as Haskell comments that are ignored by GHC but are processed by LH instead. The first phase of LH uses the Haskell compiler GHC to solve the external references, to type-check the program in the Hindley-Milner sense, and to transform it to its internal core representation. As a result, a set of type constraints are generated in the second phase, which are solved in a third phase with the help of a SMT solver. In contrast to [14], which defines constraints at the type-level, our approach specifies assertions at a specification level and uses them to statically direct a program specialization technique that produces a safe version of the input program. Then, safety checks are dynamically performed over the specialized program in the standard Maude runtime environment without resorting to external artifacts.

Also, loosely related to this work is the concept of program specialization of terminating programs based on output constraints (i.e., program post-conditions) [12]. This methodology translates the output constraints into a characterization function for the program's input that is used to guide a partial evaluation process. In contrast, we deal with non-terminating concurrent programs, and the specialization that we achieve cannot be produced by any (conventional or unconventional) partial evaluation technique for Maude programs [5].

To our knowledge, the assertion-based functionality for molding programs supported by ÁTAME is indeed beyond the capabilities of all existing Maude tools.

References

[1] Alpuente, M., Ballis, D., Frechina, F., Romero, D.: Using Conditional Trace Slicing for improving Maude Programs. Science of Computer Programming **80, Part B**, 385 – 415 (2014)

[2] Alpuente, M., Ballis, D., Frechina, F., Sapiña, J.: Assertion-based Analysis via Slicing with ABETS. Theory and Practice of Logic Programming **16**(5–6), 515–532 (2016)

[3] Alpuente, M., Ballis, D., Sapiña, J.: Inferring Safe Maude Programs with ÁTAME. In: Mathematical Software - ICMS 2018 - 6th International Conference, *Lecture Notes in Computer Science*, vol. 10931, pp. 1–10. Springer (2018)

[4] Alpuente, M., Ballis, D., Sapiña, J.: Static Correction of Maude Programs with Assertions. Tech. rep., Department of Computer Systems and Computation, Universitat Politècnica de València (2018). Available at: http://hdl.handle.net/10251/100268

[5] Alpuente, M., Cuenca-Ortega, A., Escobar, S., Meseguer, J.: Partial Evaluation of Order-sorted Equational Programs modulo Axioms. In: Proceedings of the 26th International Symposium on Logic-Based Program Synthesis and Transformation (LOPSTR 2016), *Lecture Notes in Computer Science*, vol. 10184, pp. 3–20. Springer (2016)

[6] The Anima Website (2015). Available at: http://safe-tools.dsic.upv.es/anima

[7] Clavel, M., Durán, F., Eker, S., Escobar, S., Lincoln, P., Martí-Oliet, N., Meseguer, J., Talcott, C.: Maude Manual (Version 2.7.1). Tech. rep., SRI International Computer Science Laboratory (2016). Available at: http://maude.cs.uiuc.edu/maude2-manual/

[8] Colcombet, T., Fradet, P.: Enforcing trace properties by program transformation. In: Proceedings of the 27th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '00, pp. 54–66. ACM (2000)

[9] Danvy, O., Glück, R., Thiemann, P. (eds.): Proceedings of the International Seminar on Partial Evaluation (Dagstuhl 1996), *Lecture Notes in Computer Science*, vol. 1110. Springer (1996)

[10] The *i*JULIENNE Website (2013). Available at: http://safe-tools.dsic.upv.es/iJulienne

[11] Joiner, R., Reps, T., Jha, S., Dhawan, M., Ganapathy, V.: Efficient Runtime-enforcement Techniques for Policy Weaving. In: Proceedings of the 22Nd ACM SIGSOFT International Symposium on Foundations of Software Engineering, FSE 2014, pp. 224–234. ACM (2014)

[12] Khoo, S.C., Shi, K.: Program Adaptation via Output-Constraint Specialization. Higher-Order and Symbolic Computation **17**(1), 93–128 (2004)

[13] Meseguer, J.: Conditional Rewriting Logic as a Unified Model of Concurrency. Theoretical Computer Science **96**(1), 73–155 (1992)

[14] Vazou, N., Seidel, E.L., Jhala, R.: Liquidhaskell: experience with refinement types in the real world. In: Proceedings of the 2014 ACM SIGPLAN symposium on Haskell, Gothenburg, Sweden, September 4-5, 2014, pp. 39–51 (2014)

M. Alpuente
VRAIN, Universitat Politècnica de València
Camino de Vera s/n, Apdo 22012, 46071 Valencia, Spain
e-mail: alpuente@dsic.upv.es

D. Ballis
DMIF, University of Udine,
Via delle Scienze, 206, 33100, Udine, Italy
e-mail: demis.ballis@uniud.it

J. Sapiña
VRAIN, Universitat Politècnica de València
Camino de Vera s/n, Apdo 22012, 46071 Valencia, Spain
e-mail: jsapina@dsic.upv.es